

Understanding Java Operators

1. a special symbol that can be applied to a set of variables, values, or literals—referred to as operands—and that returns a result.
2. Three flavors of operators - unary, binary, and ternary
3. Java expression is actually evaluated from right-to-left given the specific operators involved

```
int y = 4;  
double x = 3 + 2 * --y;
```

Order of operator precedence

Operator	Symbols and examples
Post-unary operators	<i>expression++</i> , <i>expression--</i>
Pre-unary operators	<i>++expression</i> , <i>--expression</i>
Other unary operators	<i>+</i> , <i>-</i> , <i>!</i>
Multiplication/Division/Modulus	<i>*</i> , <i>/</i> , <i>%</i>
Addition/Subtraction	<i>+</i> , <i>-</i>
Shift operators	<i><<</i> , <i>>></i> , <i>>>></i>
Relational operators	<i><</i> , <i>></i> , <i><=</i> , <i>>=</i> , <i>instanceof</i>
Equal to/not equal to	<i>==</i> , <i>!=</i>
Logical operators	<i>&</i> , <i>^</i> , <i> </i>
Short-circuit logical operators	<i>&&</i> , <i> </i>
Ternary operators	<i>boolean expression ? expression1 : expression2</i>
Assignment operators	<i>=</i> , <i>+=</i> , <i>-=</i> , <i>*=</i> , <i>/=</i> , <i>%=</i> , <i>&=</i> , <i>^=</i> , <i>!=</i> , <i><<=</i> , <i>>>=</i> , <i>>>>=</i>

Numeric Promotion Rules

1. If two values have different data types, Java will automatically promote one of the values
2. If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.

Numeric Promotion Rules

3. Smaller data types, namely byte, short, and char, are first promoted to int any time they're used with a Java binary arithmetic operator, even if neither of the operands is int.
4. After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

Numeric Promotion Rules

What is the data type of $x * y$?

```
int x = 1;  
long y = 33;
```

What is the data type of $x + y$?

```
double x = 39.21;  
float y = 2.1;
```

Numeric Promotion Rules

What is the data type of x / y ?

```
short x = 10;  
short y = 3;
```

What is the data type of $x * y / z$?

```
short x = 14;  
float y = 13;  
double z = 30;
```

Working with Unary Operators

Unary operator	Description
+	Indicates a number is positive, although numbers are assumed to be positive in Java unless accompanied by a negative unary operator
-	Indicates a literal number is negative or negates an expression
++	Increments a value by 1
--	Decrements a value by 1
!	Inverts a Boolean's logical value

Logical Complement and Negation Operators

The logical complement operator, `!`, flips the value of a boolean expression. For example, if the value is true, it will be converted to false, and vice versa. To illustrate this, compare the outputs of the following statements:

```
boolean x = false;  
System.out.println(x); // false  
x = !x;  
System.out.println(x); // true
```


Logical Complement and Negation Operators

Likewise, the negation operator, -, reverses the sign of a numeric expression, as shown in these statements:

```
double x = 1.21;  
System.out.println(x); // 1.21  
x = -x;  
System.out.println(x); // -1.21  
x = -x;  
System.out.println(x); // 1.21
```

Logical Complement and Negation Operators

```
int x = !5;    // DOES NOT COMPILE  
boolean y = -true; // DOES NOT COMPILE  
boolean z = !0; // DOES NOT COMPILE
```

1. The first statement will not compile due the fact that in Java you cannot perform a logical inversion of a numeric value.
2. The second statement does not compile because you cannot numerically negate a boolean value; you need to use the logical inverse operator.
3. The last statement does not compile because you cannot take the logical complement of a numeric value, nor can you assign an integer to a boolean variable.

Keep an eye out for questions that use the logical complement operator or numeric values with boolean expressions or variables. Unlike some other programming languages, in Java 1 and true are not related in any way, just as 0 and false are not related.

Increment and Decrement Operators

```
int counter = 0;  
System.out.println(counter); // Outputs 0  
System.out.println(++counter); // Outputs 1  
System.out.println(counter); // Outputs 1  
System.out.println(counter--); // Outputs 1  
System.out.println(counter); // Outputs 0
```

Assignment Operators

a binary operator that modifies, or assigns, the variable on the left-hand side of the operator, with the result of the value on the right-hand side of the equation. The simplest assignment operator is the = assignment, which you have seen already:

```
int x = 1;
```

```
int x = 1.0; // DOES NOT COMPILE
```

```
short y = 1921222; // DOES NOT COMPILE
```

```
int z = 9f; // DOES NOT COMPILE
```

```
long t = 192301398193810323; // DOES NOT COMPILE
```

Casting Primitive Values

Casting primitives is required any time you are going from a larger numerical data type to a smaller numerical data type, or converting from a floating-point number to an integral value.

```
int x = (int)1.0;  
short y = (short)1921222; // Stored as 20678  
int z = (int)9l;  
long t = 192301398193810323L;
```

```
short x = 10;  
short y = 3;  
short z = x * y; // DOES NOT COMPILE
```

Overflow and Underflow

Overflow is when a number is so large that it will no longer fit within the data type, so the system “wraps around” to the next lowest value and counts up from there. There’s also an analogous underflow, when the number is too low to fit in the data type.

For example, the following statement outputs a negative number:

```
System.out.print(2147483647+1); // -2147483648
```

Since 2147483647 is the maximum int value, adding any strictly positive value to it will cause it to wrap to the next negative number.

Compound Assignment Operators

```
int x = 2, z = 3;  
x = x * z; // Simple assignment operator  
x *= z; // Compound assignment operator
```

```
long x = 10;  
int y = 5;  
y = y * x; // DOES NOT COMPILE
```

```
long x = 10;  
int y = 5;  
y *= x;
```

```
long x = 5;  
long y = (x=3);  
System.out.println(x); // Outputs 3  
System.out.println(y); // Also, outputs 3
```


Relational Operators

<	Strictly less than
<=	Less than or equal to
>	Strictly greater than
>=	Greater than or equal to

```
int x = 10, y = 20, z = 10;  
System.out.println(x < y); // Outputs true  
System.out.println(x <= y); // Outputs true  
System.out.println(x >= z); // Outputs true  
System.out.println(x > z); // Outputs false
```

Relational Operators

`a instanceof b`

True if the reference that `a` points to is an instance of a class, subclass, or class that implements a particular interface, as named in `b`

Programming (JAVA) NC III
Marco Yimyaem

Logical Operators

x & y (AND)			x y (INCLUSIVE OR)			x ^ y (EXCLUSIVE OR)		
	y = true	y = false		y = true	y = false		y = true	y = false
x = true	true	false	x = true	true	true	x = true	false	true
x = false	false	false	x = false	true	false	x = false	true	false

Here are some tips to help remember this table:

- AND is only true if both operands are true.
- Inclusive OR is only false if both operands are false.
- Exclusive OR is only true if the operands are different.

```
boolean x = true || (y < 4);
```

Equality Operators

The equals operator `==` and not equals operator `!=` they compare two operands and return a boolean value about whether the expressions or values are equal, or not equal, respectively

The equality operators are used in one of three scenarios:

1. Comparing two numeric primitive types. If the numeric values are of different data types, the values are automatically promoted as previously described. For example, `5 == 5.00` returns true since the left side is promoted to a double.
2. Comparing two boolean values.
3. Comparing two objects, including null and String values

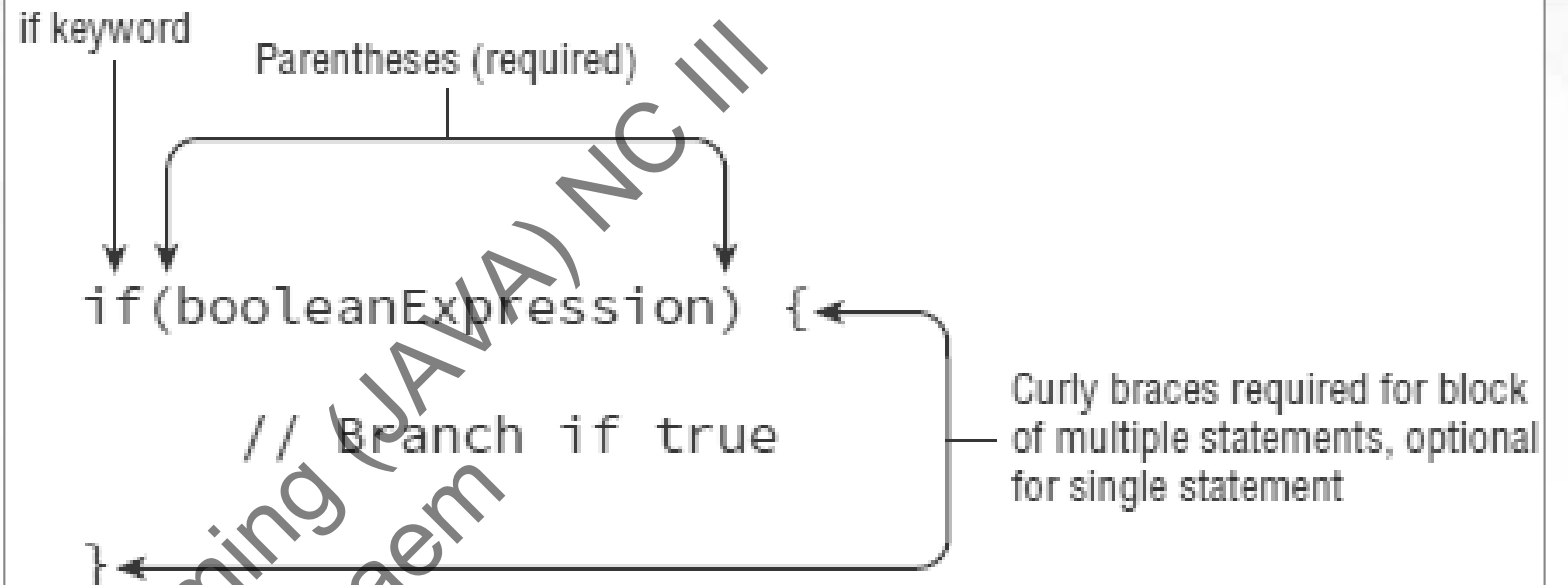
Equality Operators

```
boolean x = true == 3; // DOES NOT COMPILE  
boolean y = false != "Giraffe"; // DOES NOT COMPILE  
boolean z = 3 == "Kangaroo"; // DOES NOT COMPILE
```

```
boolean y = false;  
boolean x = (y == true);  
System.out.println(x); // Outputs true
```

```
File x = new File("myFile.txt");  
File y = new File("myFile.txt");  
File z = x;  
System.out.println(x == y); // Outputs false  
System.out.println(x == z); // Outputs true
```

The if-then Statement



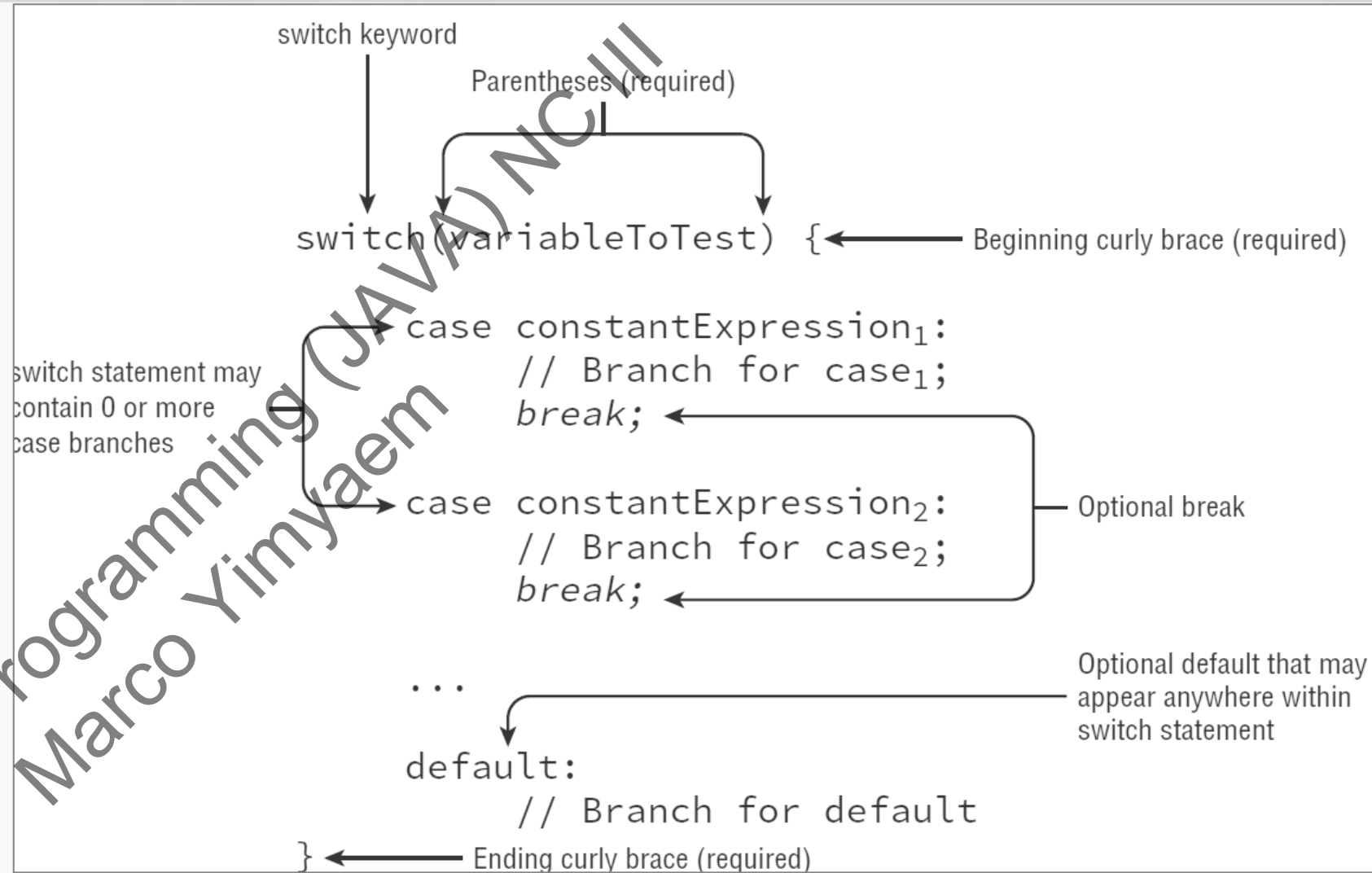
```
if(hourOfDay < 11) {  
    System.out.println("Good Morning");  
    morningGreetingCount++;  
}
```

The if-then-else Statement

```
if(hourOfDay < 11) {  
    System.out.println("Good Morning");  
} else if(hourOfDay < 15) {  
    System.out.println("Good Afternoon");  
} else {  
    System.out.println("Good Evening");  
}
```

```
if(hourOfDay < 11) {  
    System.out.println("Good Morning");  
} else {  
    System.out.println("Good Afternoon");  
}
```

The switch Statement



The switch Statement

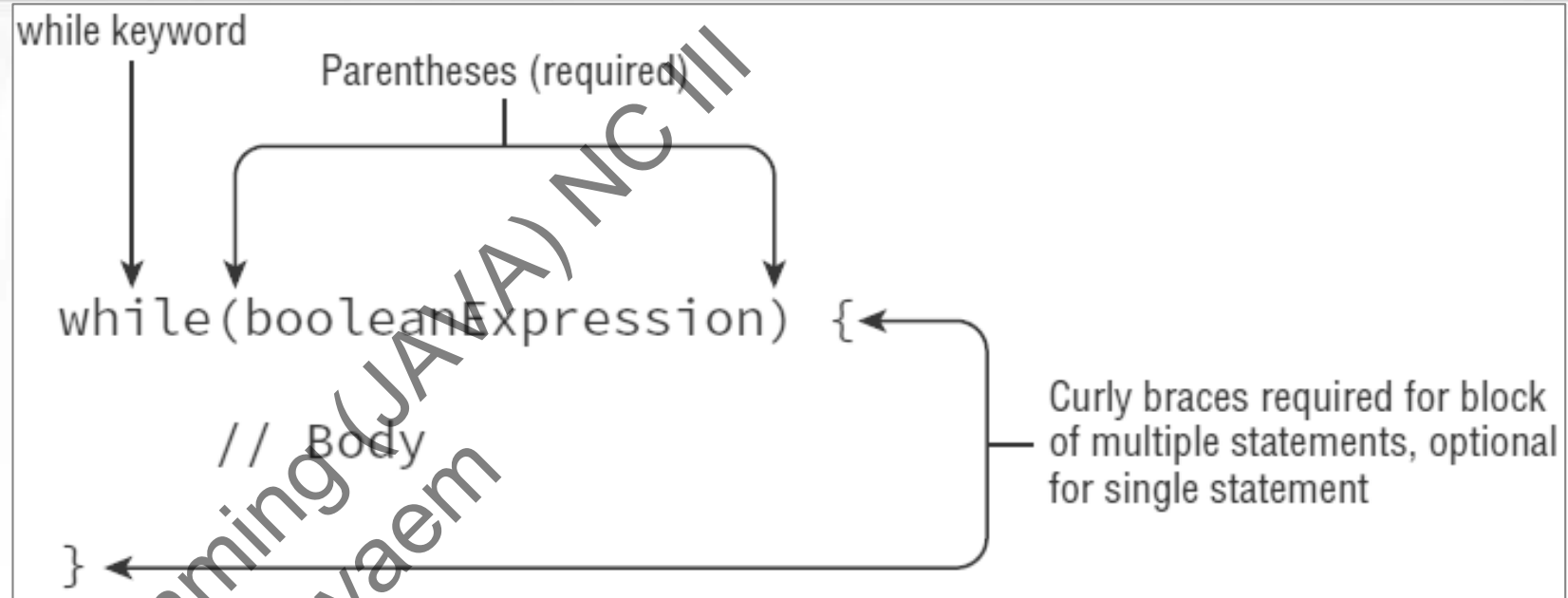
```
int dayOfWeek = 5;
switch(dayOfWeek) {
    default:
        System.out.println("Weekday");
        break;
    case 0:
        System.out.println("Sunday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
}
```

Data types supported by switch statements

- int and Integer
- byte and Byte
- short and Short
- char and Character
- String
- enum values

Note that boolean and long, and their associated wrapper classes, are not supported by switch statements

The while Statement



```
int roomInBelly = 5;
public void eatCheese(int bitesOfCheese) {
    while (bitesOfCheese > 0 && roomInBelly > 0) {
        bitesOfCheese--;
        roomInBelly--;
    }
    System.out.println(bitesOfCheese+" pieces of cheese left");
}
```

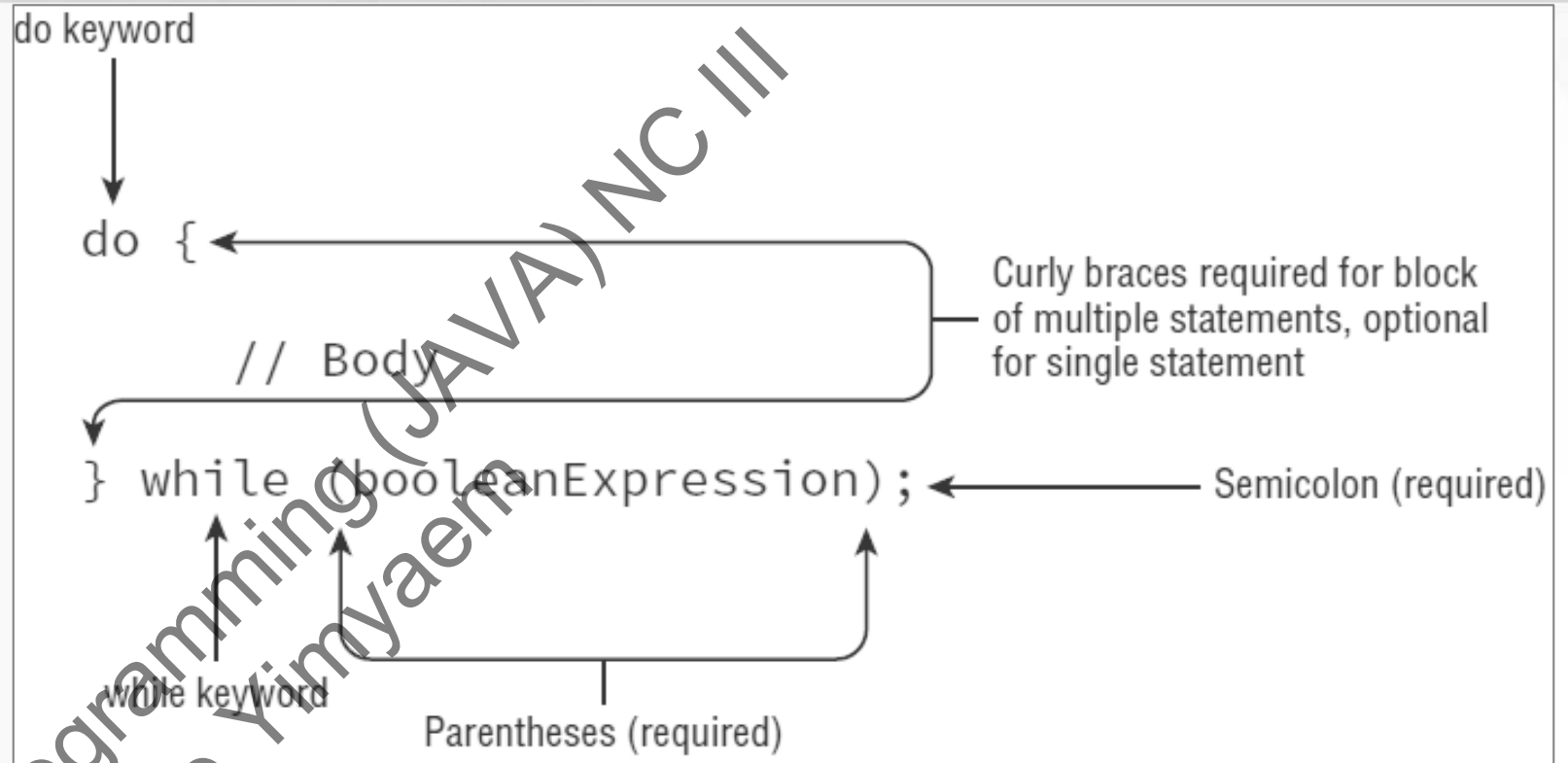
Infinite Loops

Consider the following segment of code:

```
int x = 2;  
int y = 5;  
while(x < 10)  
    y++;
```

You may notice one glaring problem with this statement: it will never end! The Boolean expression that is evaluated prior to each loop iteration is never modified, so the expression $(x < 10)$ will always evaluate to true. The result is that the loop will never end, creating what is commonly referred to as an infinite loop.

The do-while Statement



```
int x=0;
do {
    x++;
} while(false);
System.out.println(x); // Outputs 1
```

When to Use while vs. do-while Loops

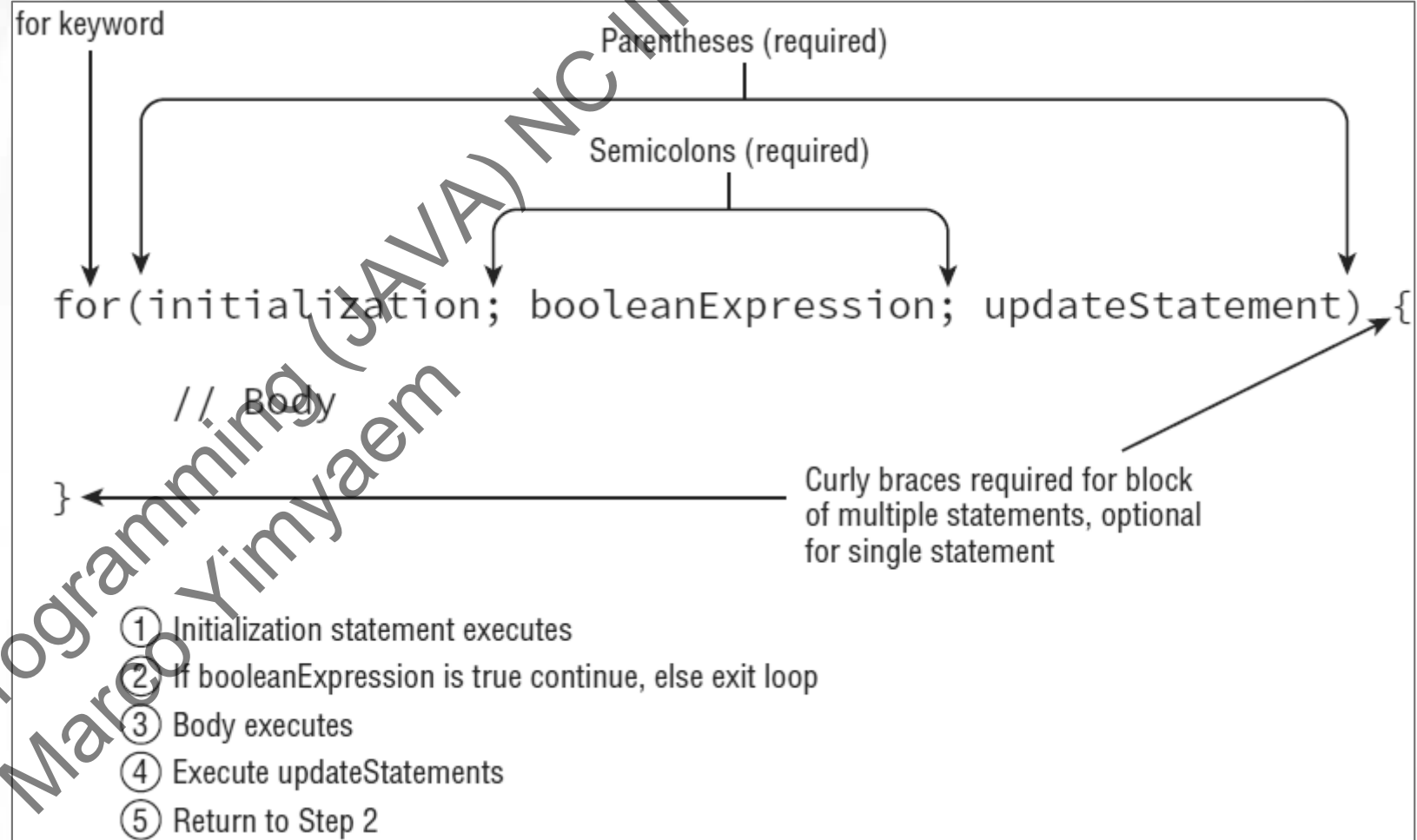
```
while(x > 10) {  
    x--;  
}  
  
and this do-while loop:  
  
if(x > 10) {  
    do {  
        x--;  
    } while(x > 10);  
}
```

Programming (JAVA) NC III
Marco Yimyaem

QnA



The for Statement





```
for(int i = 0; i < 10; i++) {  
    System.out.print(i + " ");  
}
```

Programming (JAVA) NC III
Marco Yimyaem



Creating an Infinite Loop

```
for(;;) {  
    System.out.println("Hello World");  
}
```

Programming (JAVA) NC III
Marco Yimyaem



Adding Multiple Terms to the for Statement

```
int x = 0;  
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {  
    System.out.print(y + " ");  
}  
System.out.print(x);
```



Redeclaring a Variable in the Initialization Block

```
int x = 0;  
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) { //  
DOES NOT COMPILE  
System.out.print(x + " ");  
}
```

Programming (JHIA) NC III
Marco Yimyaem



Using Incompatible Data Types in the Initialization Block

```
for(long y = 0, int x = 4; x < 5 && y<10; x++, y++) {  
    // DOES NOT COMPILE  
    System.out.print(x + " ");  
}
```

Programming JAVANC III
Marco Yimyaem



Using Loop Variables Outside the Loop

```
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {  
    System.out.print(y + " ");  
}  
System.out.print(x); // DOES NOT COMPILE
```

Programming (JAVANC III)
Marco Yimyaem



Write a program that takes in three numbers from the user and outputs the largest number.

Enter three numbers: 7 12 3

The largest number is 12

Enter three numbers: 2 2 2

All numbers are equal



Write a Java program that prompts the user to enter a positive integer n and prints the sum of the first n positive integers.

Sample Input/Output:

Enter a positive integer: 5

Sum of the first 5 positive integers: 15



Write a Java program that prompts the user to enter a positive integer n and prints the sum of the first n positive integers.

Sample Input/Output:

Enter a positive integer: 5

Sum of the first 5 positive integers: 15



Write a program that takes an integer input from the user and uses a for loop to generate the following pattern

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

Programming (JAVA)
Marco Yimyaem



Write a program that generates a multiplication table for the numbers 1 through 10

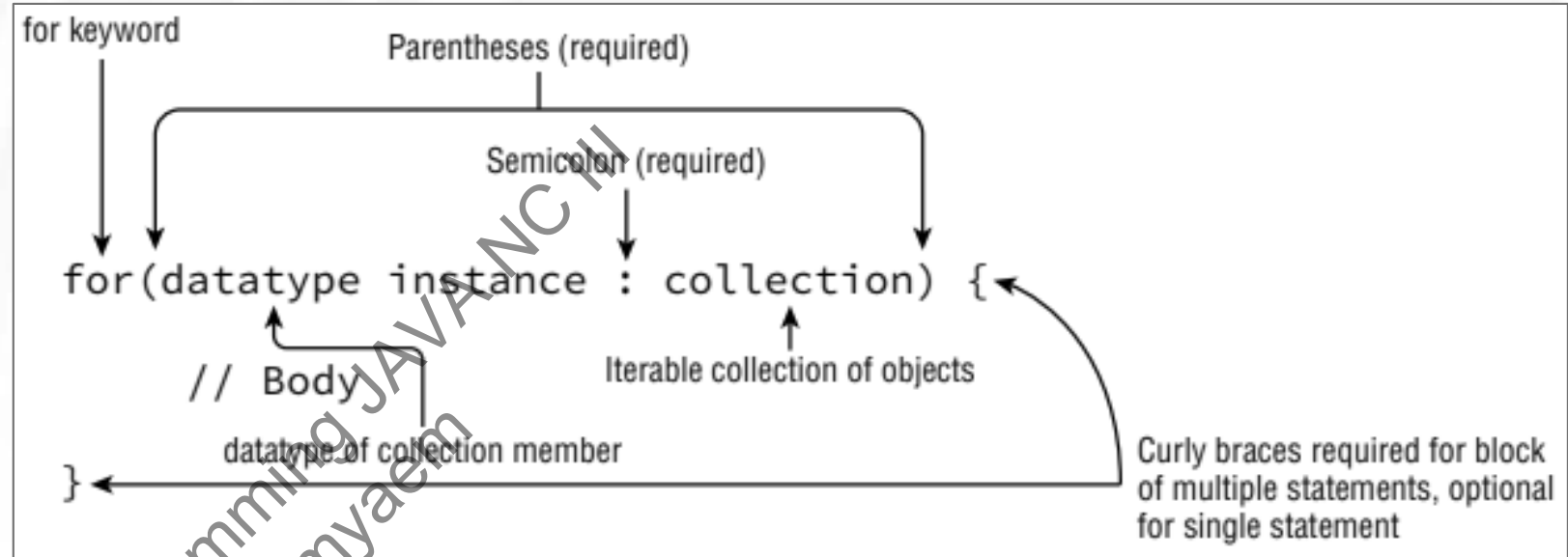
Programming (JAVA) NC III
Marco Yimyaen



Programming (JAVA) NC III
Marco Yimyaem

QnA


The for-each Statement



```
final String[] names = new String[3];
names[0] = "Lisa";
names[1] = "Kevin";
names[2] = "Roger";
for(String name : names) {
    System.out.print(name + ", ");
}
```

```
java.util.List<String> values = new java.util.ArrayList<String>();
values.add("Lisa");
values.add("Kevin");
values.add("Roger");
for(String value : values) {
    System.out.print(value + ", ");
}
```

Programming JAVA NC III
Marco Yimyaem



```
String names = "Lisa";  
for(String name : names) { // DOES NOT COMPILE  
    System.out.print(name + " ");  
}
```

```
String[] names = new String[3];  
for(int name : names) { // DOES NOT COMPILE  
    System.out.print(name + " ");  
}
```

Comparing for and for-each Loops

Since for and for-each both use the same keyword, you might be wondering how they are related. While this discussion is out of scope for the exam, let's take a moment to explore how for-each loops are converted to for loops by the compiler. When for-each was introduced in Java 5, it was added as a compile-time enhancement. This means that Java actually converts the for-each loop into a standard for loop during compilation. For example, assuming names is an array of String[] as we saw in the first example, the following two loops are equivalent

```
for(String name : names) {  
    System.out.print(name + ", ");  
}  
  
for(int i=0; i < names.length; i++) {  
    String name = names[i];  
    System.out.print(name + ", ");  
}
```


Comparing for and for-each Loops

For objects that inherit `java.lang.Iterable`, there is a different, but similar, conversion.

For example, assuming `values` is an instance of `List<Integer>`, as we saw in the second example, the following two loops are equivalent:

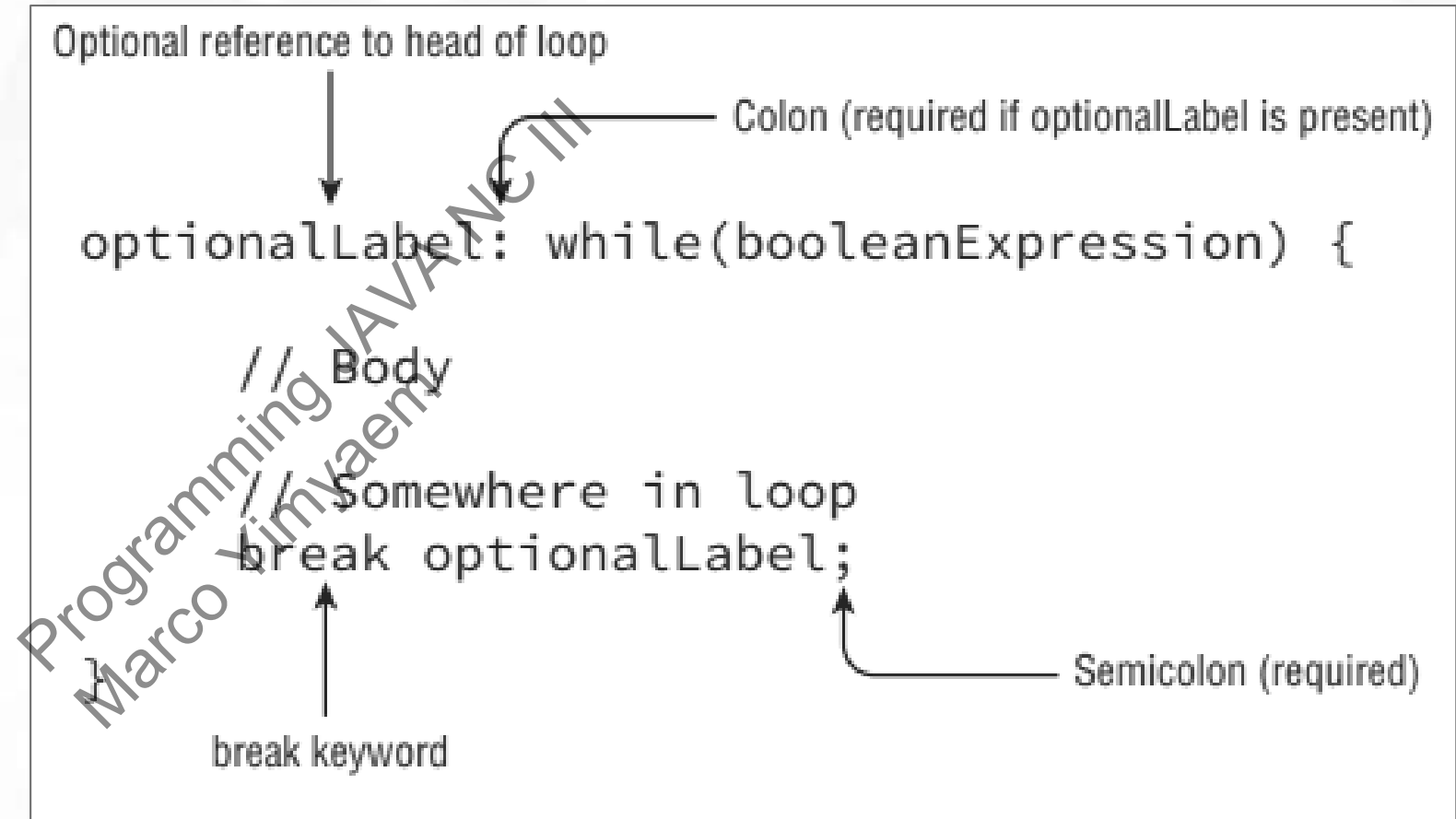
```
for(int value : values) {  
    System.out.print(value + ", ");  
}  
for(java.util.Iterator<Integer> i = values.iterator(); i.hasNext(); ) {  
    int value = i.next();  
    System.out.print(value + ", ");  
}
```

Notice that in the second version, there is no update statement as it is not required when using the `java.util.Iterator` class

Adding Optional Labels

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};  
OUTER_LOOP: for(int[] mySimpleArray : myComplexArray) {  
    INNER_LOOP: for(int i=0; i<mySimpleArray.length; i++) {  
        System.out.print(mySimpleArray[i]+"\\t");  
    }  
    System.out.println();  
}
```

The break Statement



The break Statement

```
public class SearchSample {
    public static void main(String[] args) {
        int[][] list = {{1,13,5},{1,2,5},{2,7,2}};
        int searchValue = 2;
        int positionX = -1;
        int positionY = -1;
        PARENT_LOOP: for(int i=0; i<list.length; i++) {
            for(int j=0; j<list[i].length; j++) {
                if(list[i][j]==searchValue) {
                    positionX = i;
                    positionY = j;
                    break PARENT_LOOP;
                }
            }
        }
        if(positionX== -1 || positionY== -1) {
            System.out.println("Value "+searchValue+" not found");
        } else {
            System.out.println("Value "+searchValue+" found at: " +
                "("+positionX+","+positionY+")");
        }
    }
}
```

The continue Statement

Optional reference to head of loop

Colon (required if optionalLabel is present)

```
optionalLabel: while(booleanExpression) {  
    // Body  
    // Somewhere in loop  
    continue optionalLabel;  
}
```

continue keyword

Semicolon (required)

Programming JAVANC III
Marco Yimjaem

The continue Statement

```
public class SwitchSample {  
    public static void main(String[] args) {  
        FIRST_CHAR_LOOP: for (int a = 1; a <= 4; a++) {  
            for (char x = 'a'; x <= 'c'; x++) {  
                if (a == 2 || x == 'b')  
                    continue FIRST_CHAR_LOOP;  
                System.out.print(" " + a + x);  
            }  
        }  
    }  
}
```



	Allows optional labels	Allows <i>break</i> statement	Allows <i>continue</i> statement
if	Yes *	No	No
while	Yes	Yes	Yes
do while	Yes	Yes	Yes
for	Yes	Yes	Yes
switch	Yes	Yes	No

Exam Essentials

Be able to write code that uses Java operators. This chapter covered a wide variety of operator symbols. Go back and review them several times so that you are familiar with them throughout the rest of the book. Be able to recognize which operators are associated with which data types. Some operators may be applied only to numeric primitives, some only to boolean values, and some only to objects. It is important that you notice when an operator and operand(s) are mismatched, as this issue is likely to come up in a couple of exam questions.

Understand Java operator precedence. Most Java operators you'll work with are binary, but the number of expressions is often greater than two. Therefore, you must understand the order in which Java will evaluate each operator symbol.

Be able to write code that uses parentheses to override operator precedence. You can use parentheses in your code to manually change the order of precedence.

Exam Essentials

Understand if and switch decision control statements. The if-then and switch statements come up frequently throughout the exam in questions unrelated to decision control, so make sure you fully understand these basic building blocks of Java.

Understand loop statements. Know the syntactical structure of all loops, including while, do-while, and for. Each loop has its own special properties and structures. Also, be familiar with the enhanced for-each loops that iterate over lists.

Understand how break and continue can change flow control. Know how to change the flow control within a statement by applying a break or continue command. Also know which control statements can accept break statements and which can accept continue statements. Finally, understand how these statements work inside embedded loops or switch statements.



QnA

Programming JAVA NC III
Marco Yimyaem